Architecture

Separation of Concerns

- Container Components : Handle business logic and data management
- Presentational Components : Focus purely on rendering and user interaction
- Services : Manage data operations and external API communication

Unidirectional Data Flow

Modern Angular Patterns

- Angular Signals for reactive state management
- Standalone Components for better tree-shaking
- OnPush Change Detection for optimal performance
- TypeScript-first approach with strong typing

Component Types

1. Container Components

Purpose : Handle business logic, data management, and state coordination.

Characteristics :

- Inject and use services
- Manage application state with signals
- Handle loading, error, and data states
- Coordinate between services and presentational components
- Transform data for presentation

Example Structure :

2. Presentational Components

Purpose : Focus purely on rendering and user interaction.

Characteristics :

- Receive data through inputs

- Emit events through outputs
- No direct service dependencies
- Highly reusable and testable
- Built-in accessibility features

Example Structure :

3. Services

Purpose : Handle data operations and external API communication.

Characteristics :

- CRUD operations for data management
- Reactive state with BehaviorSubject
- Comprehensive error handling
- Caching strategies
- Type-safe interfaces

Example Structure :

Data Flow Patterns

1. Data Loading Flow

2. Data Transformation Flow

3. User Interaction Flow

State Management Strategy

Signal-Based State

Reactive Updates

Component Communication Patterns

1. Parent-Child Communication

2. Service-Based Communication

Error Handling Strategy

1. Service-Level Error Handling

2. Container-Level Error Handling

3. User-Friendly Error Display

Performance Optimization

1. OnPush Change Detection

2. Signal-Based Reactivity

3. Lazy Loading

Testing Strategy

1. Presentational Component Testing

2. Container Component Testing

3. Service Testing

Best Practices

1. Component Design

- Single Responsibility : Each component has one clear purpose
- Composition over Inheritance : Use composition for reusability
- Props Down, Events Up : Data flows down, events flow up
- Pure Functions : Presentational components should be pure

2. State Management

- Single Source of Truth : Services are the source of truth
- Immutable Updates : Use signals for immutable state updates
- Computed Values : Use computed for derived state

- Error Boundaries : Handle errors at appropriate levels

3. Performance

- OnPush Change Detection : Use for all components
- Signal-Based Reactivity : Leverage Angular signals
- Lazy Loading : Load data only when needed
- Memory Management : Use takeUntilDestroyed for cleanup

4. Testing

- Unit Tests : Test components in isolation
- Integration Tests : Test component interactions
- Service Tests : Mock HTTP requests
- Accessibility Tests : Test with screen readers

Migration Guide

From Traditional Angular

- Replace Services : Update to use BehaviorSubject pattern
- Add Signals : Replace properties with signals
- Update Components : Split into container/presentational
- Add Error Handling : Implement comprehensive error handling
- Update Tests : Write tests for new patterns

From Other Patterns

- Redux to Signals : Replace Redux with signal-based state
- NgRx to Services : Use services instead of NgRx
- Smart/Dumb Pattern : Align with container/presentational pattern

Developer Checklist

Before Implementing Architecture:

- Are presentational components pure (no business logic, only rendering)?
- Do container components handle all data management and business logic?
- Are all state updates using Angular Signals?
- Is OnPush change detection enabled on all components?
- Do services use BehaviorSubject for reactive state?

- Are user interactions handled through presentational component outputs?
- Have I implemented error handling at service and component levels?
- Are data interfaces defined for component communication?
- Is takeUntilDestroyed() used for subscription cleanup?
- Does the data flow follow: Services → Containers → Presentational → User Actions?